CHAPTER

FOURTEEN

ADA

The United States Department of Defense (DoD) is a major consumer of software. Like many computer users, the Defense Department is having a software crisis. One trouble centers on the programming Babel—the department's systems are written in too many different languages. This problem is particularly acute for applications involving embedded systems—computers that are part of larger, noncomputer systems, such as the computers in the navigation systems of aircraft. Since timing and machine dependence are often critical in embedded systems, programs for such systems are often baroque and idiosyncratic. Concerned about the proliferation of assembly and programming languages in embedded systems, the DoD decided in 1974 that it wanted all future programs for these systems written in a single language. It began an effort to develop a standard language.*

Typical embedded systems include several communicating computers. These systems must provide real-time response; they need to react to events as they are happening. It is inappropriate for an aircraft navigational system to deduce how to avoid a mountain three minutes after the crash (in the unlikely event that the on-board computers are still functioning three minutes after the crash). A programming language for embedded systems must include mechanisms to refer to the duration of an event and to interrupt the system if a response has been delayed. Thus, primary requirements are facilities for exception handling, multi- and distributed processing, and real-time control. Since the standard is

* Fisher [Fisher 78] and Carlson [Carlson 81] describe the history and motivation of that project in greater detail.

a programming language, the usual other slogans of modern software engineering apply. That is, the language must support the writing of programs that are reliable, easily modified, efficient, machine-independent, and formally describable. A request for proposals produced 15 preliminary language designs. The Defense Department chose four of these for further development. After a two-year competition, it selected a winner. This language was christened "Ada" in honor of Ada Augusta, Countess of Lovelace, a co-worker of Babbage and the first programmer.

Ada was created in the limelight. Many members of the academic and industrial computer science community contributed advice and criticism to the development process. The result is a language whose scope is ambitious. *SIGPLAN Notices* served as a forum for much of the debate surrounding the specification and development process.

Ada is at the far language end of the language-model spectrum.* The entire syntax and most of the formal semantics of Ada have been specified [Donzeau-Gouge 80]. The language is progressing towards standardization [DoD 80]. Its conceptual basis and the foundation of its syntax are derived from Pascal, a language renowned for its simplicity. However, the designers of Ada, in trying to satisfy the numerous requirements of the Ada specification, created an extensive and complicated language.

This section describes the Ada facilities for distributed processing and communication. We do not consider all the intricacies of Ada, since a complete description of Ada would itself fill a book.† Jean Ichbiah led the group at CII-Honeywell-Bull that designed Ada. Several Ada compilers have been completed and the DoD has great expectations for Ada's eventual widespread application. For those who wonder about the effect of U.S. military support on the popularity of a programming language, the last language promoted by the Defense Department was Cobol.

Entry, Access, and Rendezvous

Explicit processes and synchronized communication are the basis for concurrency in Ada. In many ways, Ada builds on concepts from Distributed Processes. Ada borrows Distributed Processes's remote procedure call and extends it in three important ways. (1) The entry procedures of Distributed Processes become objects (entries) with properties such as size and order, accessible from many places

* We characterize models as being a simple description of distributed computing, unadorned by syntax, and languages as embedding (and perhaps obscuring) their ideas for distribution in the practical aspects of programming. By that metric, Ada is at the far, far language-end of the language-model spectrum.

† Many books devoted to describing Ada have already been published. One such book is Pyle's *The Ada Programming Language* [Pyle 81]. Similarly, a "self-assessment procedure" in the *Communications of the ACM* was devoted to a tutorial on the nonconcurrent aspects of Ada [Wegner 81].

within a single process. (2) Called processes are not passive—they schedule the order in which they serve their entries. (3) Though calling processes are, by and large, passive, they can abort calls if they do not receive a quick enough response.

Ada is a complete programming language. Its designers intended to provide the programmer with a useful set of facilities (such as process communication, queueing, and abstract data types) while still permitting manipulation of the system primitives (such as interrupt locations, processing failures, and queue sizes). The overall effect is a language that is frightening in complexity but impressive in scope.

Processes in Ada are called *tasks*. Tasks have local storage and local procedures. Ada tasks are objects. The programmer can declare a single instance of a particular task or describe a *task type*, generating instances of that type much as one would create new instances of a Pascal record. Our first few examples deal with individual declarations of tasks. If a subprogram declares three tasks, then the tasks are created when that subprogram is entered. When created, a task commences processing. Tasks can terminate in several different ways, such as reaching the end of their program, being explicitly stopped, or by causing a run-time error. Ada also has a mechanism for synchronizing the termination of a collection of tasks. We discuss synchronized task termination at the end of this chapter.

Ada permits arbitrary nesting of program descriptions. A task can declare other tasks, which in turn can declare still other tasks. Additionally, Ada procedures can be recursive. Thus, a recursive Ada procedure that declares a new task creates a new instance of that task for each recursive call. Tasks can also be created by explicit execution of the task-creation command. Thus, Ada has both explicit process creation and process creation through lexical elaboration. A task that creates other tasks is the *parent* of these tasks; these tasks are *dependent* on the parent and are *siblings*. Thus, if task P creates tasks Q and R, P is the parent of Q and R while Q and R are siblings.

One of Ada's design goals is *encapsulation*—hiding the implementation of a subsystem while exhibiting (to other program segments) the subsystem interface. This intention is realized by breaking the description of a task into two parts, a specification and a body. The *specification* is visible to the other program components. That is, at compilation other components can use the information in the specification. The specification describes the names and formats of the interfaces to this task from other tasks. The *body* contains the task's variable declarations and code. It is hidden from other tasks and subprograms. That is, other program segments cannot reference the internal structure and state described in a task body. The separation of a task into specification and body syntactically enforces intermodule security and protection.

Communication requires syntactic connection—names for mutual reference. We create a communication channel to a task by declaring an *entry* in the task's specification. Syntactically, other tasks treat that entry as a procedure. Within the called task the entry represents a queue of requests. A statement of the form

accept <entry name> <formal parameter list>; is a directive to retrieve the next call from an entry's queue and to process it with the code that follows. The structure of the information exchange is similar to a procedure call: there are named and typed fields for information flow both into and out of the task.*

Our first example is a task, line_block, that assembles lines of 120 characters and forwards them for printing. A line is

```
type line is array (1 .. 120) of character;
```

When a line is full, line_block passes it to the printer task. The specification part of line_block defines its name and declares a communication channel, entry add. The parameters in this declaration define the shape of communications to this channel, not particular identifiers for actual processing.

```
task line_block is
    entry add (c: in character);
end line_block;
```

To place the character "d" in the line being assembled, another task would execute the command

```
line_block.add ("d");
```

The declaration of an entry creates a queue and calls on that entry are placed in that queue. But for an exception discussed below, a task that calls an entry blocks until that call is handled.

Line_block invokes **accept** on an entry to get the next item in a queue. The **accept** call supplies a formal parameter list. Thus, in the *module* (hidden, invisible) body of line_block, a statement of the form

```
accept add (c: in character);
```

takes the next item from entry add and assigns the value of the calling argument to variable c. The scope of this variable is the **accept** statement (discussed below). The completion of the **accept** unblocks the calling task; it resumes processing. If there are no calls waiting in the entry queue, then the **accept** statement blocks until one arrives. Variable c is an **in** parameter because it channels information *into* the task.

Line_block first accumulates a line of 120 characters. It then requests that the line be printed by calling entry writeit in task printer. It repeats this process for successive lines. The **task body** of line_block is as follows:

* Unfortunately, the Ada documentation is deliberately ambiguous about the semantics of parameter passing. Evidently, particular Ada implementations can use either call-by-value-result or, when feasible, call-by-reference for intertask communication.

```
task body line_block is
    thisline : line;
    i        : integer;
begin
    loop
        for i in 1 .. 120 loop
            accept add (c: in character) do     -- The scope of c is the accept
                                                   statement.
                thisline (i) := c;     -- In Ada, semicolons are statement
                                          terminators, not separators.
            end add;     -- Block structure is usually indicated by "<keyword>
                            ... end <identifier>" pairs, instead of "begin ...
                            end" pairs. We usually pick this identifier to be the
                            keyword that began the block or, for accept
                            statements, the name of the accept entry.
        end loop;
        printer.writeit (thisline);
    end loop;
end line_block;
```

Ada provides **out** parameters for communicating responses back to a calling task. To illustrate **out** parameters, we extend line_block to respond to character insertions with a count of the character positions remaining on the line. This requires two modifications to line_block. The first is to include an **out** (result) parameter in entry add. This changes both the specification part of the task and the **accept** statement. The second is to add a critical region after the **accept** statement. During this critical region, the calling and called task are synchronized and the called task computes its response. The calling task blocks until after the critical region. Syntactically, the critical region is the sequence **do** <statements> **end**; following the **accept**. The called process returns the value of the **out** parameter at the end of the critical region. The time between the execution of the **accept** and the **end** of the accept statement is called a *rendezvous* between the calling and the called tasks.

```
task line_block is
    entry add (c: in character; left: out integer);
end line_block;

task body line_block is
    thisline : line;
    i        : integer;
begin
    loop
        for i in 1 .. 120 loop
```

```
            accept add (c: in character; j: out integer) do    -- rendezvous
                j := 120 − i;    -- the critical region
            end add;             -- end of rendezvous
            thisline (i) := c;
        end loop;
        printer.writeit (thisline);
    end loop;
end line_block;
```

Indeterminacy Timing, delays, and time-outs are important for real-time systems. Languages for embedded systems need mechanisms to deal with time. In Ada, a process can execute a **delay** command to suspend processing for a specified interval. For example, if the value of current is five, executing the statement

```
delay 2*current;
```

causes this task to pause for ten seconds. As we shall see, delay is also an integral part of the Ada communication mechanism.

CSP uses guarded commands to select a process that is ready to communicate from among several possible communicators. Ada's **select** statement generalizes CSP's guarded input command, allowing other alternatives besides blocking until communication. **Select** takes a sequence of select alternatives, separated by the delimiter **or**. Each alternative is an accept alternative, a delay alternative, or a terminate alternative. *Accept alternatives* attempt to read a value from an entry. Delay and terminate alternatives are used when the system cannot immediately accept an input. A *delay alternative* sets an alarm clock. If the alarm goes off before an acceptable request arrives, the task executes the code of the delay alternative instead of accepting. If an acceptable request arrives first, the alarm is disabled and the request accepted. *Terminate alternatives* are used to bring a set of tasks to simultaneous conclusion. We discuss terminate alternatives below. Both accept and delay alternatives can be followed by a series of statements to be executed when that alternative is selected.

Like guarded commands, select alternatives can be conditional on a boolean expression. An alternative of the form **when** b => **accept** e can be selected only if boolean condition b is true. A select alternative is *open* when it has either no guard or a true guard.

In line_block, we might prefer to separate the mechanisms for adding a character to the output line and returning a count of the available character positions. This requires that line_block have two entries, an add entry to add a character and a free entry to request the free space count.

```
task line_block is
    entry add (c: in character);
    entry free (left: out integer);
end line_block;
```

```
task body line_block is
    thisline : line;
    i        : integer;
begin
    loop
        i := 1;
        while i < 121 loop
            select
                accept add (c: in character) do
                                -- Selection of this alternative first executes
                                   the accept statement and then increments
                                   the counter.
                    thisline (i) := c;
                end add;
                i := i + 1;
            or
                accept free (j: out integer) do
                    j := 120 - i;
                end free;
            end select;
        end loop;
        printer.writeit (thisline);
    end loop;
end line_block;
```

Sometimes we prefer that a task not block if there are no pending requests. An *else alternative* provides this possibility. Syntactically, an else alternative substitutes **else** for the select statement's last **or** and follows the **else** with a series of statements. Semantically, a select statement with an else alternative that cannot immediately accept an entry request executes the statements of the else alternative instead.

Ada has several syntactic restrictions on the arrangement of select alternatives. Every select statement must have at least one accept alternative. If it has a terminate alternative, it cannot have a delay alternative; if it has a terminate alternative or a delay alternative, it cannot have an else alternative.

The syntax of the select statement allows the expression of a variety of control structures. Therefore, the algorithm followed in evaluating a select statement is somewhat complex. Its theme is to select an immediately available, open accept alternative. If no such alternative exists, then the arrangement of waiting, delay, termination, and else alternatives determines the task's behavior. More particularly, evaluation of a select statement proceeds as follows: (1) The task checks the guard of each alternative and discovers the open alternatives. Each open alternative is an accept, delay, or terminate alternative. (2) It determines to which entry each open accept alternative refers. (As we discuss

below, entries can be subscripted. Determining the referent entry is equivalent to computing the entry subscript.) We call entries with open accept alternatives and waiting requests in the entry queue the *acceptable* entries. (3) The task determines how long a delay each delay alternative specifies. (4) If there are any acceptable entries, the task executes the action associated with one of them (selecting one arbitrarily). That is, the first choice of a select statement is always to immediately accept an entry call. (5) If there are no acceptable requests waiting, then we might want to wait for one, wait for one but give up after a while, execute some alternative action, or check if it is time to coordinate the termination of a set of tasks. (6) If the select statement has no delay alternatives, no terminate alternative, and no else alternative, then the task waits until a request appears in an acceptable entry queue. This is typical behavior for a passive, "server" task. (7) If the select statement has an else alternative, the process executes that alternative immediately. (8) If the select statement has a delay alternative, the task blocks. It unblocks after the shortest delay. It then executes the statement associated with that delay. If an acceptable request appears before the delay has elapsed, the task executes an accept alternative that refers to that request instead. (9) If the select statement has a terminate alternative, then the terminate alternative may be executed under certain circumstances.

Of course, there may be no open alternatives. In that case, if the select statement has an else alternative, it evaluates that alternative. A select statement with neither open alternatives nor an else alternative has reached an error. It raises the **select_error** exception. Figure 14-1 graphs the decision flow of the select statement.

In our next example, we vary line_block to illustrate the select statement. We assume that line_block no longer forces lines out to the printer, but waits for the printer to ask for them. (Thus, line_block becomes a producer-consumer buffer.) Furthermore, if the internal buffers are full, a line is ready for the printer, and the printer fails to request that line within 5 minutes (300 seconds), line_block calls the routine printer_trouble. We provide line_block with two buffers, one for filling with incoming characters and another to hold a line ready to be sent to the printer. The task has variables that record the state of these buffers: nextfree, the next free character position in the filling buffer; and print_ready, a boolean that is true when the printing buffer is ready to be written.

```
task line_block is
    entry add          (c: in character);
    entry please_print (ln: out line);
end line_block;

task body line_block is
    printer_trouble_time : constant integer := 300;
    print_line, fill_line    : line;    -- a buffer for the printer and a buffer for
                                           filling
```

```
    nextfree              : integer;
    print_ready           : boolean;    -- Is the printing line ready to output?
begin
    nextfree    := 1;
    print_ready := false;
    loop
        select
            when (nextfree < 121) =>    -- space for another character
                accept add (c: in character) do
                    fill_line (nextfree) := c;
                end add;
                nextfree := nextfree + 1;
                if (nextfree = 121) and not print_ready then
                    print_ready := true;
                    nextfree    := 1;
                    print_line  := fill_line;    -- By subscripting our two
                end if;                             buffers, we could have
                                                    avoided this copying.
        or
            when print_ready =>    -- full buffer ready for the printer
                accept please_print (ln: out line) do
                    ln := print_line;
                end please_print;
                if nextfree = 121 then
                    print_line := fill_line;
                    nextfree   := 1;
                else
                    print_ready := false;
                end if;
        or
            when print_ready and (nextfree > 120) =>
                delay printer_trouble_time;
                    printer_trouble;    -- waiting for the printer for over five
        end select;                        minutes
    end loop;
end line_block;
```

Ada programs can reference attributes of certain objects. An *attribute* of an object is non-value information about that object. For example, one attribute of an entry is the size of its queue. Syntactically, an attribute of an object is the name of the object, an apostrophe, and the attribute name. For example, the attribute add'count is the size of the add entry queue; line_block'terminated is true when task line_block has terminated.

Ada entries can be subscripted, producing a *family* of entries. If line_block is a buffer for 20 tasks, each with its own printer, then line_block would declare

Figure 14-1 Selection statement operation.

a family of entries for each character producer and each printer. We must successively poll the entries to find one with a waiting request. (This contrasts with CSP, where we can check an entire family of processes in a single guarded input command.)

```
task line_block is
    entry add (1..20) (c: in character);     -- a family of entries
    entry please_print (1..20) (ln: out line);
end line_block;
```

```
task body line_block is
    buffers  : array (1..20) of line;     -- no double buffering this time
    bufnext : array (1..20) of integer;-- pointer into each buffer
    thisone  : integer;                   -- entry currently under consideration

begin
    for thisone in 1..20                  -- initialize the buffers
        loop
            bufnext (thisone) := 1;
        end loop;
    thisone := 1;

    loop    -- This loop polls each entry pair to see if that producer/printer
            pair is ready to interact. If nothing is waiting in the appropriate
            entry, we select the else alternative and proceed to poll the next
            pair.
        select
            when bufnext (thisone) < 121 =>     -- space for another
                                                   character in this buffer
                accept add (thisone) (c: in character) do
                    buffers (thisone) (bufnext (thisone)) := c;
                end add;
                bufnext (thisone) := bufnext (thisone) + 1;
        or
            when bufnext (thisone) > 120 =>     -- full buffer ready for the
                                                   printer
                accept please_print (thisone) (ln: out line) do
                    ln := buffers (thisone);
                end please_print;
                bufnext (thisone) := 1;
        else
            null;    -- If neither is ready, go on to next member of the family.
        end select;
        thisone := thisone + 1;
        if thisone > 20 then thisone := 1; end if;
    end loop;
end line_block;
```

We could specify 20 printers to handle the 20 calls with

```
task printer (1..20) is
end printer;
```

The printers are numbered from 1 to 20. Within its task body, a printer can reference its own number as the attribute printer'index.

Task types Tasks can be types, just as records, arrays, and enumerations are types. We can have both statically allocated tasks (such as arrays of tasks) and dynamically created tasks. Processes execute the command **new** to dynamically create new instances of a task type. Of course, one needs pointers to dynamically created objects. Ada calls such pointers *access types*.

We illustrate task types with a program for the dining philosophers problem. The dining philosophers program has three varieties of tasks: philosophers, forks, and the room. Philosophers call the room to enter and exit, and call the forks to pick them up and put them down. Philosophers cycle through thinking, entering the room, picking up the forks, eating, putting down the forks, and leaving the room. To declare a new type, task type fork, we state

```
task type fork is
    entry pickup;
    entry putdown;
end task;
```

A pointer to a fork is an afork.

```
type afork is access fork;
```

Philosophers are also a task type. In our example, we create them dynamically and then send them pointers to each of their forks.

```
task type philosopher is
    entry initiate (left, right: in afork);    -- This is a template for the structure
                                               -- of the initiate entry, not a
                                               -- declaration of left and right.
end task;
```

We give ourselves a fixed, initial room.

```
task room is
    entry enter;
    entry exit;
end task;
```

The task bodies of the processes are as follows:

```
task body room is
    occupancy: integer;
begin
    occupancy := 0;
    loop
        select
            when (occupancy < 4) =>
```

```
                accept enter;
                occupancy := occupancy + 1;
        or
            accept exit;
            occupancy := occupancy − 1;
        end select;
    end loop;
end task;
task body fork is
begin
    loop
        accept pickup;
        accept putdown;
    end loop;
end task;
```

New does not provide creation parameters for the newly generated object, so we call the initiate entry in the new philosopher to send it the names of its forks. Since the entry parameters (leftparm and rightparm) last only through the scope of the accept statement, we need permanent variables left and right to remember the names of the forks.

```
task body philosopher is
    left, right: afork;
begin
    accept initiate (leftparm, rightparm: in afork) do
        left  := leftparm;
        right := rightparm;
    end;
    loop
        -- think;
        room.enter;
        left.pickup;
        right.pickup;
        -- eat;
        left.putdown;
        right.putdown;
        room.exit;
    end loop;
end task;
```

The entire program is as follows:

```
procedure dining_ps is pragma main;    -- In Ada, the "pragma" (compiler
                                          advice) "main" asserts that this is
                                          the main program.
```

```
task room is ... ;
task body room is ... ;
task type fork is ... ;
task body fork is ... ;
type afork is ... ;
task type philosopher is ... ;
task body philosopher is ... ;

philos    : array (0..4) of philosopher;     -- declarations of global storage
i         : integer;
theforks : array (0..4) of afork;
begin
     for i in 0..4 loop
          theforks (i) := new fork;
     end loop;
     for i in 0..4 loop                      -- Send each philosopher its forks.
          philos (i).initiate (theforks (i), theforks ((i + 1) mod 5));
     end loop;
end dining_ps;
```

This solution avoids both deadlock and starvation. The room's occupancy limit (four philosophers) prevents deadlock and the fork entry queues prevent starvation.

Selective entry call In Ada, an accepting task has some control over ordering the processing of calls to its entries. It can select the next entry for processing, choose the first arrival from among several entries, abort a potential rendezvous if a time constraint is exceeded, and even use the size of its entry queues in deciding what to do. Calling tasks do not have an equivalent variety of mechanisms to control communication. However, Ada does provide calling processes with a way of aborting calls that "take too long." This mechanism is the select/entry call. A *select/entry call* takes one of two forms—either a conditional entry call or a timed entry call. The form of a *conditional entry call* is

```
select
     <entry call> (<parameters>); s1; s2; ... ;
else
     s'1; s'2; ... ;
end select;
```

A *timed entry call* is similar

```
select
     <entry call> (<parameters>); s1; s2; ... ;
```

```
or
    delay <time-expression>; s'1; s'2; ... ;
end select;
```

In a conditional entry call, a rendezvous takes place if the called task is waiting for a request on this entry. After the rendezvous, the calling task executes statements s_1; s_2; .... If the called task is not waiting for a request on this entry, the calling task executes the **else** statements s'_1; s'_2; .... In either case, this task does not block for long; it either communicates immediately or does not communicate at all. The timed entry call is similar, except that the call aborts if the rendezvous does not begin before the end of the indicated delay. If the delay is exceeded, the call is abandoned and the task executes statements s'_1; s'_2; ....

Unlike the multiple arms of the select/accept statement, a calling task can offer to communicate with only a single other task in any select/entry call. This avoids the potential difficulty of matching several possible communicators. A task executing a conditional entry call offers the called task an opportunity for rendezvous. Either the called task accepts immediately or the calling task withdraws the offer. This organization allows a simple protocol in which only two messages need be sent to accept or reject a rendezvous offer.

The timed entry call is somewhat more complex; its semantics is complicated by the issue of whose clock (which task) is timing the delay. Bernstein discusses the more elaborate protocols involved in many-to-many communication matching in his paper on output guards in CSP [Bernstein 80]. He argues that matching many-to-many requests requires complicated or inherently unfair protocols. Since even the timed entry call allows only many-to-one offers, it is simpler to program the protocols of Ada than of CSP output guards.

Since requests can be withdrawn from an entry queue, programmers cannot (in general) treat a nonzero count as assuring the existence of a call. That is, the statement

```
if thisentry'count > 0 then
    accept thisentry ...
end if;
```

can block if the request to thisentry is withdrawn between the test of the **if** statement and the **accept**.

Ada treats interrupts as hardware-generated entry calls. It provides *representation specifications*, a mechanism for tying specific interrupts to particular entries. A task can disable an interrupt by ignoring requests on that interrupt's entry.

The elevator controller Ada was designed for programming embedded systems. As an example of an embedded system, we present a decentralized elevator control system. This system schedules the movements of several elevators. Due

not so much to the wisdom of building decentralized elevator controllers as much as to our desire to illustrate distributed control, many components in this system are processes. More specifically, a task controls each elevator and a task controls each button (up and down) on each floor. An elevator task controls moving an elevator up and down, opening and closing its doors, and stopping at floors. Each floor button task waits for its button to be pressed, then signals the next approaching elevator to stop. Figure 14-2 shows the communication structure of the elevator system.

```
procedure elevator_controller is
pragma main
begin
    basement      : constant integer := 0;    -- elevators for a building of
    penthouse     : constant integer := 40;   -- 40 stories
    num_elevators : constant integer := 8;    -- There are eight elevators.
    floor_wait    : constant integer := 15;   -- Elevators stop at a floor for
                                                 (at least) 15 seconds.

  -- The motor procedure accepts commands of the form "up," "down," and
     "stop." A direction (the ways an elevator can move) is a subtype of a
     motor_command, either "up" or "down." A floor is an integer from
     basement (0) through penthouse (40).

    type motor_command is (up, down, stop);
    subtype direction is motor_command range up .. down;
    type floor is range basement .. penthouse;
```

A floor button task has four entries. The press entry is for the line from the real (physical) button. Each time the real floor is pressed, a call is added to this entry's queue. When an elevator approaches a floor from a particular direction, it calls that floor's button for that direction on the coming entry. That is, an upward moving elevator arriving at a floor calls that floor's up button task. This call is a select/entry call. If the button is waiting on coming (someone has pressed it and no other elevator has promised to come) then the tasks rendezvous. The elevator detects the rendezvous and knows that it is sought. (The elevator also stops at floors requested by its internal buttons, the car_buttons.) When an elevator arrives at a floor it announces its arrival with a call to the here entry. The button also has an interested entry for communicating with idle elevators (described below). Floor buttons never initiate communication. Instead, they wait for elevators to call them. Additionally, all communication is between anonymous elevators and floor buttons. No elevator ever communicates directly with another elevator and no floor button knows which elevator will serve it.

Sometimes elevators find themselves with no pressing demands. An elevator with none of its car_buttons on *dreams*—that is, it surveys floors until it finds

Figure 14-2 The elevator system.

one that needs its services. This survey is done without actually moving the elevator. To distinguish between elevators that are actually going to a floor and the ones that are just considering going, elevators conduct this survey on the interested entry. A floor with waiting passengers responds to a single interested call. However, if another elevator indicates that it can get to that floor first by signaling the floor on the coming line, the floor accepts the offer. Thus, an interested connection is a promise by an elevator to go to a floor but entails no commitment by that floor to save its passengers for the elevator.

```
task type button is
    entry press;
    entry coming;
    entry interested;
    entry here;
end task;

task body button is
    press_on, coming_on, interested_on: boolean;
```

```
   -- When an elevator arrives at a floor, the button on that floor calls the
      "arrived" procedure. This procedure clears the button's press queue and
      delays the departure of the elevator until at least one second after the
      last press. Thus, a passenger can keep an elevator at a floor by holding
      down the button on that floor.

   procedure arrived is
   begin
      delay floor_wait;
      loop
         select
            accept press;
            delay 1;              -- time for another press
         else
            exit                  -- exit the loop
         end select;
      end loop;
      press_on      := false;
      coming_on     := false;
      interested_on := false;
   end arrived;
begin                             -- the main procedure of task button
   loop
      select
         accept here do           -- always respond to arrivals
            arrived;
         end here;
      or
         when not press_on =>
            accept press;         -- Check if the button has been pressed.
            press_on := true;
      or
         when press_on and not coming_on =>
            accept coming;        -- Accept the first "coming" if the
            coming_on := true;       button has been pushed.
      or
         when press_on and not coming_on
                  and not interested_on =>
            accept interested;    -- Accept at most one expression of
            interested_on := true;   interest, and only if no elevator has
                                     promised to come.
      end select;
   end loop;
end button;
```

```
   -- This statement generates 2·(penthouse−basement+1) buttons and sets
      them running. (In practice, we might omit the "down" button for the
      basement and the "up" button for the penthouse.)
buttons: array (floor, direction) of button;
```

Each elevator controller is a task with two entries. The first is for requests from its own internal floor selection buttons, the car_buttons. The second is from the floormark_reader. When the elevator is moving, the floormark_reader interrupts as each floor approaches, signaling the floor's number. This signal allows enough time to decide whether to stop the elevator at that floor.

```
task type controller is
    entry car_buttons (f: in floor);
    entry floormark_reader (f: in floor);
end task;
```

The controller procedures motor and door_move cause the starting and stopping of the elevator and the opening and closing of the elevator doors. The motor procedure takes the commands up, down, and stop; the door procedure takes open and close. The machine hardware ensures that the motor never drives the elevator through the foundation or roof. Stop stops the elevator at the next floor. We ignore the internal structures of these procedures, other than the minor caveats implied in their comments.

```
task body controller is
    type doormove is (open, close);
  -- Motor returns from a stop call when the floor is reached and the
     elevator has stopped. Motor returns from an up or down immediately.
    procedure motor (which_way: in direction) is
                    ⋮

  -- This procedure opens and shuts the doors.
    procedure doors (how: in doormove) is
                    ⋮
```

The elevator's permanent state is stored in three variables. The current_floor is the elevator's current floor. The current_direction is the elevator's current (or intended) direction. The array goingto (a two-dimensional, boolean array, indexed by floors and directions) is the elevator's set of intended visits. A ⟨floor, direction⟩ entry is set in goingto when the elevator promises to visit that floor with a coming call or when an elevator passenger requests that floor. Variables found and where are temporaries.

```
type setting is array (floor, direction) of boolean;
goingto           : setting;
current_floor     : floor;
current_direction : direction;
where             : floor;
found             : boolean;
```

Three simple auxiliary functions on floors and directions are step, limit, and opp_direction. Function step takes a floor and a direction, and returns the next floor in that direction. Thus, a step(up) from floor 5 is floor 6. A step(down) from floor 5 is floor 4. The limit of a direction is the farthest floor in that direction. For example, limit(down) = basement. Function opp_direction reverses directions. The opp_direction(up) = down. For the sake of brevity, we omit the code for these functions.

Initializing the elevator controller consists of moving it to the basement and setting the elements in its goingto array to false.

```
procedure initialize (aset: in out setting;
                      cur_floor: out floor;
                      cur_dir: out direction) is
    f   : floor;
    dir : direction;
begin
    doors (close);
    motor (down);
    loop
        select
            accept floormark_reader (flr: in floor) do
                f := flr;
            end floormark_reader;
            exit when f = basement;
        or
            delay 60;     -- If no new floor mark has shown up in the last
            exit;            minute, we must be at the bottom.
        end select;
    end loop;
    for f in basement .. penthouse
    loop
        for dir in up .. down
        loop
            aset (f, dir) := false;
        end loop;
    end loop;
```

```
    cur_floor := basement;
    cur_dir   := up;
end initialize;
```

Procedure check_carbuttons reads the floors requested from entry car_buttons and sets the appropriate values in array goingto.

```
procedure check_carbuttons (cur_floor: in floor;
                            goingto: in out setting) is
begin
    cbs: loop
        select
            accept car_buttons (f: in floor) do
                if f > cur_floor then
                    goingto (f, up) := true;
                else
                if f < cur_floor then
                    goingto (f, down) := true;
                end if;
            end car_buttons;
        else
            exit cbs;    -- We can exit a labeled loop.
        end select;
    end loop;
end check_carbuttons;
```

An elevator calls procedure arrive_at_floor when it reaches a floor. The elevator does not leave until the floor button returns from the call to here. Since the floor button empties its press entry queue during the here rendezvous, one can keep an elevator at a floor by repeatedly pressing the call button on that floor.

```
procedure arrive_at_floor (cur_floor: in floor; cur_dir: in direction) is
begin
    doors (open);
    buttons (cur_floor, cur_dir).here;
    goingto (cur_floor, cur_dir) := false;
    doors (close);
end arrive_at_floor;
```

Function further_this_way is true when an element of goingto, "further along in this direction," is set. Additionally, further_this_way enforces promises found by dreaming (see below) by ensuring that the elevator goes at least as far as the get_to floor.

```
function further_this_way (cur: in floor;
```

```
                        dir: in direction;
                        get_to: in floor) returns boolean is
    answer : boolean;
    f      : floor;
begin
    answer := false;
    if not (cur = limit(dir)) then
        f := step(cur, dir);
        loop
            answer:= goingto(f, dir) or (get_to = f);
            exit when answer or (f = limit(dir));
            f := step(f, dir);
        end loop;
        return answer;
    end if;
end further_this_way;
```

The system workhorse is function move_til_stop. It takes the elevator as far as needed in a given direction, calling the procedures that run the motor and open and close the doors.

```
function move_til_stop (dir: in direction; get_to: in floor) returns floor is
    cur: floor;
begin
    motor (dir);              -- An elevator continues until told to stop.
    mainloop: loop
        accept floormark (f: in floor) do
            cur := f;
        end floormark;
        select                -- entry call select
            buttons (cur, dir).coming;
            goingto (cur, dir) := true;
        else                  -- attempt only an immediate rendezvous
            null;
        end select;
        check_carbuttons (cur, goingto);
        if goingto (cur, dir) then
            motor (stop);
            arrive_at_floor (cur, dir);
            exit mainloop
                when not further_this_way (cur, dir, get_to);
            motor (dir);
        end if;
    end loop;
```

```
    return cur;
end move_til_stop;
```

A difficult part of elevator control is deciding what to do when an elevator does not have a currently pending request, such as a pressed car_button. When that happens, our elevators *dream*—pretend to move up and down the building, looking for a floor that is interested in having an elevator visit. A floor that accepts an interested call is promised that this elevator will move to that floor. However, the floor can still respond to a coming call from another elevator if the other elevator gets there first.

```
procedure dream (start, stop, realfloor: in floor;
                 dir: in direction;
                 answer: out boolean;
                 finds: out floor) is
begin
    check_carbuttons (realfloor, goingto);
    answer := false;
    finds  := start;
    loop
        select
            buttons (finds, dir).interested;
            answer := true;
        else
            answer := goingto (finds, dir);
        end select;
        if finds = stop or answer then
            exit;
        end if;
        finds := step (finds, dir);
    end loop;
end dream;

- - - - - - - - - - - - - - - - - - - - - main program - - - - - - - - - - - - - - - - - - -
begin
    initialize (goingto, current_floor, current_direction);

  -- First dream of going in the current direction until the limit (basement
     or penthouse) of that direction, then in the opposite direction down from
     that limit to the other limit, and so forth, until a real request appears.
     Then satisfy requests until at a limit. Then go back to dreaming of work.
    loop
        dream (current_floor, limit (current_direction), current_floor,
               current_direction, found, where);
```

```
            if found then
                current_floor := move_til_stop (current_direction, where);
            else
                dream (limit (current_direction), current_floor, current_floor,
                        opp_direction (current_direction), found, where);
                if found then
                    current_floor := move_til_stop (current_direction, where);
                else
                    current_direction := opp_direction (current_direction);
                end if;
            end if;
        end loop;
    end controller;

    -- Create a controller for each elevator.
    vators: array (1 .. num_elevators) of controller;

end elevator_controller;
```

In this example, we periodically call procedure check_carbuttons to examine the elevator's internal buttons. This could be a task running concurrently with the main program. Ideally, such a task would share the array goingto with the controller. In Ada, one can share storage between tasks—tasks created in the same scope share variables declared in that scope and higher scopes. Exercise 14-7 asks that the elevator controller be modified to have a process that checks car-buttons execute in parallel with the process that controls elevator movement.

Pragmatics

Packages Several of Ada's features are of particular pragmatic interest. The first is the inclusion in the language of packages, a form of abstract data type facility. The second is the ability to declare generic program objects. Generic objects reflect Alan Perlis's maxim that "one man's constant is another man's variable" [Perlis 82]. They allow the type information in Pascal-like languages to be instantiated to different values. For example, Pascal requires separate sorting routines to sort integers, reals, and so forth. In Ada, one could build a generic sorting routine and instantiate that routine to particular data types.

Priorities A task may be given an integer-valued priority. A task of higher priority has greater urgency. Thus, a disk-controller task would typically have a higher priority than a terminal-controller task. In a task specification, the pragma

pragma priority <compile-time integer expression>

associates a priority with a task. The Ada standard states the intended effects of a priority as [DoD 80, p. 9-13]:

> If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same processing resources then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.

The standard goes on to warn [DoD 80, p. 9-13]:

> The priority of a task is static and therefore fixed. Priorities should be used only to indicate relative degrees of urgency; they should not be used for task synchronization.

Attribute taskname′priority gives the priority of task taskname.

Dynamic exception handling In Ada, certain run-time incidents are *exceptions*. Ada subprograms can have a section of code reserved to deal with each particular variety of exception. A program segment that runs when an exception happens is an *exception handler*. Typical exceptions include numeric exceptions, raised on conditions such as underflow and overflow; select exceptions, raised when a select statement without an else alternative has no open alternatives; and tasking exceptions, raised when intertask communication fails. Programmers can declare their own exceptions. The command raise <exception-name> raises an exception, forcing control to that exception's handler. If an exception occurs and the program segment has no exception handler, the exception propagates back through the run-time calling structure of the task until a handler for it is found. If an exception propagates to the outermost level of a task without finding an exception handler, the task terminates.

Termination An Ada task terminates when it has reached the end of its code and all its dependent tasks have terminated. This is *normal termination*. Selection of a terminate alternative in a select/access statement also causes normal termination (see below). Execution of an **abort** statement causes *abnormal termination*. Any task may abort any task; a task is not limited to aborting only itself or its dependent tasks. Abnormal termination of a task causes abnormal termination of its dependent tasks. Thus, if a system of tasks is floundering, the entire system can be terminated by aborting the parent task. The attribute taskname′terminated is true if task taskname has terminated; the attribute taskname′completed is true if task taskname is waiting for the termination of its dependent tasks before terminating.

A task that terminates while waiting in an entry queue is deleted from that queue. If rendezvous is in progress when a calling task terminates, the called task completes the rendezvous normally. If a task calls an entry in a task that has terminated, the tasking_error exception is raised in the calling task. The termination of the called task during rendezvous also raises the tasking_error exception in the calling task.

The terminate alternative of the select/accept statement is designed to coordinate the termination of a set of tasks. The idea is that a task may wish to terminate when it is no longer possible for it to receive any further entry calls. This can be the case only if its parent and all its sibling and dependent tasks are either terminated or in this same "potentially dying" state. Algorithmically, we imagine a task that is active as flying a white flag. When a task terminates, it lowers the white flag and raises a black one. A task waiting in a select statement with an open terminate alternative flies a gray flag. If and only if all other tasks that can call the gray-flag task have dark (gray or black) flags does this task terminate (changing gray to black). If a task flying a gray flag receives a call to one of its entries, it changes its flag to white and continues processing. A mostly dark landscape is not an assurance that termination is near—one white flag can eventually cause a sea of gray flags all to turn white. To simplify determining which tasks can still potentially receive communications, creating tasks with the **new** statement precludes using a terminate alternative.

Perspective

Ada is an imperative, explicit process language that provides synchronized communication. Communication is a form of remote procedure call. However, unlike procedure calls, the called process keeps multiple entries and the calling process can abort the communication for inadequate service.

Ada is an attempt to deal with the issues of real multiple-processor, distributed computing systems. The facilities for synchronization provided by rendezvous, multiple entries, and the temporal constructs provide Ada great operational power. Ada provides a well-developed set of mechanisms for dealing with temporal relationships, such as elapsed time and time-outs. This variety of mechanisms is not surprising; concepts such as delay and time-out are important for manipulating objects in the real world. By and large, other languages omit these functions because they complicate the language semantics. Explicit time (much like multiple processes) introduces an element of indeterminacy into a programming system. For most programming tasks this indeterminacy is a hindrance to the writing of correct programs.

Ada has been criticized for the asymmetry of its communication relationships. Calling and called tasks are not equal, though it is not clear where the balance of power lies. A calling task knows with which task it is communicating. However, it can only select its communicators serially, with little more control over the occasion of communication than time delay.* Called tasks can schedule their work with much greater flexibility, choosing indeterminately as requests arrive. However, calls to a task are anonymous. This ignores a potential form of interprocess security.

* Even this power, embodied in the select/entry call, was a late addition to Ada—it is not in the preliminary manual [SIGPLAN 79], only the later standard [DoD 80].

In providing mechanisms for handling process and communication failure, Ada moves beyond the simpler proposals. In so doing, it has limited the distributed aspects of the language. The visibility of attributes such as task'terminated and the presence of terminate alternatives in select statements does not mean that an Ada program cannot be implemented on a distributed system; only that there is more underlying sharing of information than may be apparent at first glance.

Ada tries to deal with the pragmatic issues in programming. On the other hand, it tries to be all encompassing, to provide a mechanism for every eventuality. This produces a language that is not only powerful but also complex. Ada has been criticized for this complexity (for example, by Hoare in his Turing Award lecture [Hoare 81]). But Ada is also to be praised for its scope and depth. Its future is difficult to predict: It has the potential to soar on the power of its mechanisms, or to become mired in a morass of syntactic and semantic complexity.

PROBLEMS

14-1 Imagine an alarm clock process in an Ada system that can keep time (and, of course, use all the other process communication mechanisms). Can this process be used to replace the delay alternative in the select statement?

14-2 Kieburtz and Silberschatz [Kieburtz 79] cite the example of a "continuous display" system of two processes, one that calculates the current position of an object (the update process) and the other that (repeatedly) displays the object's location on a screen (the display process). The two processes run asynchronously; typically, the update process is faster than the display process. The goal of the system is to display the latest position of the object; when update calculates a new value for the position of the object, the old ones become useless. However, the display process should never be kept waiting for an update. Thus, the information from update should not be treated by display with the usual first-in-first-out (queue) discipline. Instead, it is the most recently received value that is needed. Old values are useless and should be discarded.

This idea of display/update interaction is isomorphic to an organization that shares storage between the two processes. Of course, shared storage is easy to arrange in Ada. This question requests a program for the continuous display problem that does not rely on shared storage.

14-3 To what extent can the effect of **select**/**accept** be achieved by checking the size of the entry queue with the **count** attribute? What are the pitfalls of this approach?

14-4 Contrast the **select**/**accept** and **select**/**entry** call mechanisms with Exchange Functions (Chapter 7) and CSP's guarded input and output commands (Chapter 10).

14-5 Procedure check_carbuttons sets an element in goingto for floors f less than or greater than the current floor. It does not set an element when f equals cur_floor. Why?

14-6 Modify the elevator controller program to turn on and off the lights on the up and down buttons on each floor.

14-7 Program an "elevator button checking task" that runs concurrently with the elevator control and communicates with it by sharing the array goingto.

14-8 Can a passenger on floor 2 be ignored by elevators that are kept busy between floors 5 and 8? (*Clossman*)

14-9 What happens when two elevators arrive at a floor at the same time?

14-10 Improve the program of the elevator controller to run the elevators more efficiently. For example, have a floor serviced by another elevator send a cancellation message to an interested elevator.

14-11 Devise an algorithm that mimics the effect of terminate alternatives without using terminate alternatives.

REFERENCES

[**Bernstein 80**] Bernstein, A. J., "Output Guards and Nondeterminism in 'Communicating Sequential Processes,'" *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 2 (April 1980), pp. 234–238. This paper discusses the protocol difficulties in dealing with processes that can issue guarded input and output commands. Ada sidesteps much of this problem by allowing only a single destination for a select/entry call.

[**Carlson 81**] Carlson, W. E., "Ada: A Promising Beginning," *Comput.*, vol. 14, no. 6 (June 1981), pp. 13–15. Carlson's paper is a brief history of the Ada development effort. He combines this history with predictions about Ada's future.

[**DoD 80**] Department of Defense, "Military Standard Ada Programming Language," Report MIL-STD-1815, Naval Publications and Forms Center, Philadelphia, Pennsylvania (December 1980). This is the current Ada standard. This document will be replaced by future standards as Ada evolves. The standard is about 200 detailed pages long. This is a good measure of Ada's complexity.

[**Donzeau-Gouge 80**] Donzeau-Gouge, V., G. Kahn, and B. Lang, "Formal Definition of the Ada Programming Language: Preliminary Version for Public Review," unnumbered technical report, INRIA (November 1980). This paper is a formal definition of all aspects of Ada except tasking. The paper presents two kinds of semantics for Ada, "static semantics" and "dynamic semantics." The static semantics performs type checking and the like. The dynamic semantics expresses the run-time semantics of programs in an "applicative subset" of Ada.

[**Fisher 78**] Fisher, D. A., "DoD's Common Programming Language Effort," *Comput.*, vol. 11, no. 3 (March 1978), pp. 24–33. This article relates the motivations for and historical development of Ada.

[**Hoare 81**] Hoare, C.A.R., "The Emperor's Old Clothes," *CACM*, vol. 24, no. 2 (March 1981), pp. 75–83. This paper was Hoare's Turing Award lecture. In this paper he warns about the pitfalls of programming languages that are too complicated.

[**Kieburtz 79**] Kieburtz, R. B., and A. Silberschatz, "Comments on 'Communicating Sequential Processes,'" *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2 (January 1979), pp. 218–225. Kieburtz and Silberschatz's paper is the source of the "continuous display" problem.

[**Perlis 82**] Perlis, A. J., "Epigrams on Programming," *SIGPLAN Not.*, vol. 17, no. 9 (September 1982), pp. 7–13. Perlis presents a satirical collection of programming wisdom.

[**Pyle 81**] Pyle, I. C., *The Ada Programming Language*, Prentice-Hall International, Englewood Cliffs, New Jersey (1981). This book is a good introduction to the complexities of Ada for the experienced programmer. It is both concise and comprehensive. Particularly useful are appendices on Ada for programmers familiar with Fortran or Pascal.

[**SIGPLAN 79**] SIGPLAN Notices, "Ada Manual and Rationale," *SIGPLAN Not.*, vol. 14, no. 6 (June 1979). This is the original report from the Honeywell Ada group on their language. This report was published as a two volume issue of SIGPLAN Notices and is widely available. The current Ada manual [DoD 80] supersedes it as the Ada standard. SIGPLAN Notices published much of the discussion and many of the proposals that led to Ada. ACM now has a technical group, ADATEC (a "junior" special interest group) devoted to Ada.

[**Wegner 81**] Wegner, P., "Self-Assessment Procedure VIII," *CACM*, vol. 24, no. 10 (October 1981), pp. 647–677. This paper presents a self-assessment procedure on Ada. Because of Ada's novelty, Wegner attempts not only to test, but also to teach the language. He concentrates on the abstraction aspects of Ada, such as modules, types, and packages. He completely excludes discussion of concurrency in Ada.